DESIGN AND IMPLEMENTATION OF PROPHESY AUTOMATIC INSTRUMENTATION AND DATA ENTRY SYSTEM

Xingfu Wu Valerie Taylor
Department of Electrical and Computer Engineering, Northwestern University, Evanston IL 60208
Email: {wuxf, taylor}@ece.northwestern.edu

Rick Stevens
Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439

ABSTRACT

In this paper, we present the Prophesy Automatic Instrumentation and Data Entry (PAIDE) system and describe its design framework and implementation supporting C, Fortran77 and Fortran90 programs on diverse systems. We use NAS Conjugate Gradient (CG) and Integer Sort (IS) benchmarks to analyze the PAIDE's instrumentation overheads for two granularities of instrumentation with increasing number of processors and problem size. The experimental results on the SGI Origin2000 show that the instrumentation overhead for CG benchmarks is less than 3.4%, while that for parallel IS benchmarks is less than 1%. This is significant and it indicates that the PAIDE system does not perturb the performance data. The PAIDE system is able to capture performance information on repeated events as sequential events and is appropriate for parallel and distributed applications that take a long time to execute.

KEYWORDS: Source-code-level instrumentation, automatic instrumentation, performance database, performance data collection and entry.

1. INTRODUCTION

Traditionally, to tune algorithm implementations for best performance, detailed performance data must be gathered during actual executions of a program on a given target architecture. Instrumentation is the mechanism used to gather data from the executing program. Software instrumentation involves the placement of small pieces of code in a compiler, a parallel communication library, an application program, or even the linked executable. The function of this code, referred to as an event code, is to report some values to the components responsible for aggregating the event data. The level at which the instrumentation is placed determines the types of displays which can be created. This paper presents the Prophesy Automatic Instrumentation and Data Entry (PAIDE) system, for which the goals are minimize instrumentation overhead and collect enough performance data so as to generate detailed performance models.

A way to implement an instrumentation system entails modifying an application source code before compile-time to insert calls to collect performance data. The instrumentation of an application source code can provide information about abstract high-level events and constructs within the source code. For example, Pablo [10] and SvPablo [11] compose an automatic/interactive instrumentation system. They use a preprocessor (SvPabloParser) to read the source code in order to produce an annotated source code with instrumentation calls. Another approach is to directly instrument the application source code during the compile time. For example, CONVEX [5] uses a modified compiler to insert instrumentation at the desired location. This provides access to information that is only available to compilers, such as data and loop dependencies, etc.

Instrumentation of the parallel communication libraries provides an easy way to collect data about the interactions between processes. It can provide trace information consisting of send and receive events as well as computation and communication statistics. An advantage of this approach is that it does not require the programmers to modify their programs, just re-link them with special flags. For example, IPS-2 [9] does not require any modifications of the user's program; its instrumentation is automatically inserted at compile/link time. The MPI Standard (Message Passing Interface) [3] specifies a standard profiling interface that allows tool developers to easily attach their instrumentation instructions without the need to have access to the MPI source code [3, 6]. For PVM (Parallel Virtual Machine) library [2], the trace data is generated by instrumented versions of the PVM routines. The instrumented routines, apart from performing the application message operations, use PVM to send event tracing messages to a monitoring process. Another approach is to use binary rewriting technique to directly insert instrumentation into the executable image. For example, MTOOL [4] uses the technique. The binary instrumentation permits data collected in a language-independent way, and makes it

easy to collect information about the parallel communication libraries in the same way that application data is collected. It does not require any source code. Paradyn [7, 8] uses dynamic instrumentation technique to modify an executable's instructions to generate performance data. It does not require any source code yet.

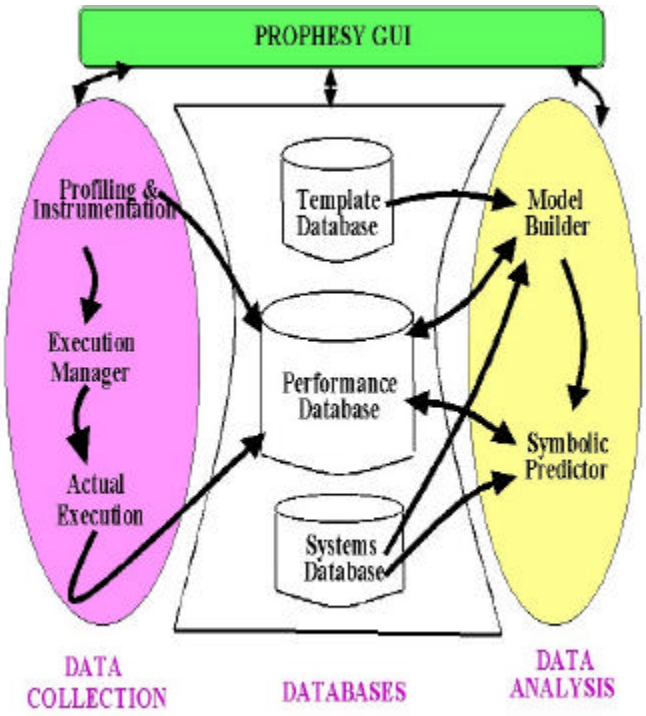

Figure 1. Prophesy framework

The Prophesy system [13, 12] is an infrastructure for analyzing and modeling the performance of parallel and distributed applications. It targets a number of programming models, languages and architectures. The Prophesy system framework consists of three major components: data collection, data analysis, and three central databases, as illustrated in Figure 1. The data collection component focuses on the automatic instrumentation and application code analysis at the level: basic blocks, procedures, or functions. The entire code can be instrumented at the basic block level such that a significant amount of performance information can be gathered to gain insight into the performance relationship between the application and hardware and between the application, compilers, and run-time systems. The resultant performance data will be stored in the performance database organized by basic blocks, functions, and modules. The data analysis component can produce an analytical performance model with coefficients, at the granularity specified by the user. The models are developed based upon performance data from the performance database, model templates from the template database, and system characteristics from the systems database. An application goes through three stages (instrumentation of the application, performance data collection of many runs, and model development using optimization techniques) to generate an analytical performance model. Prophesy system allows for the development of linear as well as nonlinear models. These models, when combined with data from the system database, can be used by the prediction engine to predict the performance on a different compute platform.

The use of databases with Prophesy system allows users to explore the performance models developed for different kernels, applications and systems. The data in the databases are organized in a hierarchical manner, allowing for the development of analytical models of different granularities. Prophesy system is an infrastructure designed to explore the plausibility and credibility of various techniques in performance evaluation (such as scalability, efficiency, speedup, performance coupling between application kernels, etc.) and allow users to use various metrics collectively to bring performance analysis environments to the most advanced level. The Prophesy interface uses web technology to enable users from anywhere to access the performance data, add performance data, or utilize the automated instrumentation and modeling processes.

The PAIDE system is the data collection component of Prophesy system, with the goal of minimizing instrumentation overhead and code. The PAIDE inserts minimal instrumentation into the source code to identify high-level problems such as too much synchronization blocking and to generate high-level detailed performance models. The PAIDE system is able to capture information on repeated as well as single events; information is collected to generate first and second order statistics on the various code segments. There is no I/O during the execution; only one I/O write is completed at the end of the execution. Tools such as SvPablo [11] etc. that require I/O during execution result in significant overhead. It was determined that such tools can result overhead for the NAS Conjugate Gradient (CG) and Integer Sort (IS) benchmarks [1] of 30% or more. In contrast, the PAIDE system resulted in overheads of less than 3.4% for CG and less than 1% for parallel IS. This is significant and indicates that the PAIDE does not perturb the performance data.

The rest of this paper is organized as follows. Section 2 presents the detailed design framework of the PAIDE system and its implementation. Section 3 uses NAS CG and IS benchmarks to do performance analysis and comparison. Section 4 summaries the paper.

2. FRAMEWORK OF THE PAIDE SYSTEM

The PAIDE provides a way to do automatic instrumentation for Fortran77, Fortran90, and C programs. Its basic organization is illustrated in Figure 2. The Parser shown in Figure 2 entails inserting some instrumentation codes to a source file. It provides the following options:

-ALL: Instrumenting all procedures and outer loops
-PROC: Instrumenting all procedures

-NOP: Instrumenting all procedures not nested in loops
-LOOP: Instrumenting all loops
-FTP: Using anonymous ftp to automatically transfer some required files to the Prophesy server

While a source code is instrumented via the Parser, its performance relation file and call graph file are also generated. If the Parser with "-FTP" option is used, the two files are automatically sent to the Prophesy server via anonymous ftp for entry into the Prophesy database. The performance relation file is generated according to event ID order. For each event, the event information includes: event ID, event start line number, event end line number, procedure name, caller name, caller file (module name). The Call graph shows the call graph relations for source codes. The instrumented source code can be compiled to generate an instrumented executable. At the end of the program execution, each process generates a per-process performance file. The performance files, one for each processor are automatically sent to the Prophesy server via anonymously ftp. Then, the corresponding Perl CGI scripts are used to automatically put the data into Prophesy database via Prophesy web interface.

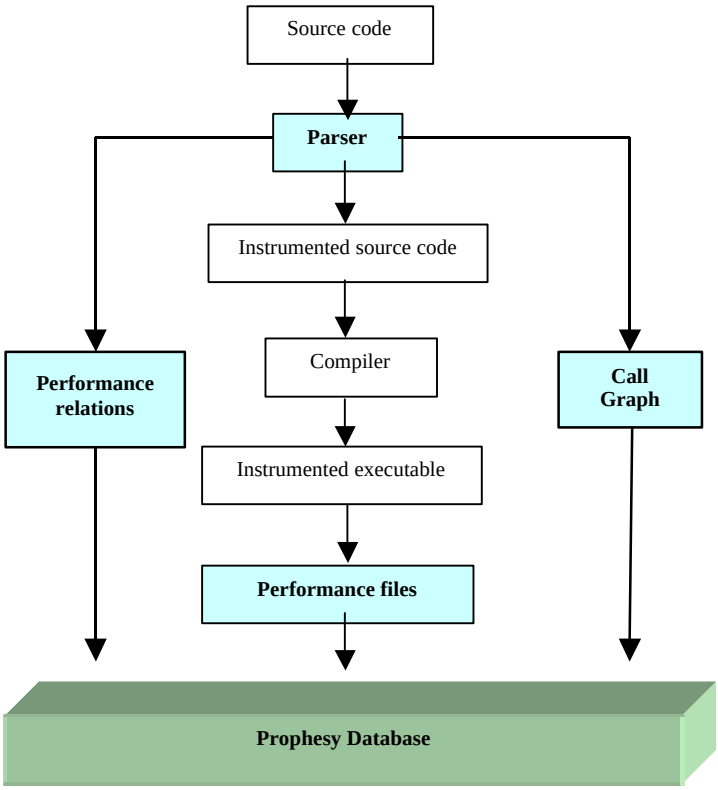

Figure 2. Basic organization of the PAIDE system

To achieve our goal of minimal instrumentation code, the PAIDE generates statistics in-line for coarse-grain as well as fine-grain computation. The details of the instrumentation code are described as follows.

2.1 Performance Data

Basically, parallel computing involves three types: computation, communication and I/O. For computation, its metrics of interest may be count and duration. Communication includes send (source) and receive (destination) operations. For send (source) operation, its metrics of interest may be count, start time, duration and message size. For receive (destination) operations, its metrics of interest may be duration and message size. Communication includes the communication between processors, or between processor and memory etc. For I/O operation, its metrics of interest may be count, duration and data size. The PAIDE collects data on all three types. Further, performance statistics are collected for different levels of granularities. Since a basic unit can be accessed from multiple functions, statistics are collected at the levels of both the function and basic unit. The basic unit and function statistics are differentiated by access. This is accomplished by keeping up with the calling routine and associating a different event identifier with the calling routine. For example if function C is called by functions A and B, the execution of function C has two event identifiers, one for each of the two functions A and B.

Based on the analysis as above, we insert the following types of instrumentation codes: counter, timer and square of each timer. The counter and timer are used to generate the mean of the execution times. The square of each timer is used to generate the standard deviation. Generally, counters count the frequency of an event while timers measure the time spent using a resource of the system. Counter and timer instructions can be inserted at appropriate locations in the application program by the PAIDE system.

In Figure 2, when a source code is instrumented via the Parser, the Parser counts each procedure and loop in the source code, and labels it with a unique instrumentation point number (or called event identifier). The basic PAIDE instrumentation structure is defined as follows:

```
struct PAIDE {
        long    count;      /* call count */
        double  runtime;    /* duration */
        double  psqrtime;   /* sum of square of duration */
};
struct PAIDE celero[Total number of instrumentation points];
```

2.2 Automatic Instrumentation

The PAIDE provides multiple instrumentation options described as above so that the user can choose to instrument only procedures or loops, the procedures not nested in loops and outer loops, or all procedures and outer loops. The user also can select the source files to be instrumented. But the main file containing the "main()" function for a C program or the "PROGRAM" subroutine for a Fortran program must be instrumented. The default instrumentation option is to instrument all procedures and all loops for all source files.

The PAIDE takes a source code to generate the corresponding instrumented file. If the source code contains the main routine, it renames the C main function to "*_Prophesy_main_*()", or changes the Fortran main routine PROGRAM to SUBROUTINE. It also generates an extra initial main file "XX_init.c" containing a new "main()". In the instrumented Fortran code, the original main routine like "PROGRAM NAME" is substituted with "SUBROUTINE NAME" and "XX_init.c" contains a call to NAME_ or NAME (based on different C/Fortran call interfaces). Note that, in Fortran90, the module/array has the same syntax as function calls like YY(…). If instrumenting the function calls is allowed, the PAIDE will automatically instrument all the function calls plus the data structures with form of YY(…). If the program is large, and has a lot of modules, there will be a lot of code unnecessarily instrumented. Therefore, for Fotran77 and Fortran90 programs, the PAIDE only instrument subroutines. In the instrumented C code, the original main function "main()" is replaced with "*_Prophesy_main_*()", and "XX_init.c" contains a call to "*_Prophesy_main_*()".

For example, given NAS Parallel IS benchmark "is.c", after it is instrumented, the following files are generated: the initial main file "is_init.c", the instrumented source code "is_inst.c", the performance relation file "eventlist.is", and the call graph file "callgraph.is". An example of the in-line instrumentation code is given below:

```
/*  This is the main iteration of is.c */
#line 952 "Instrumentation code"
prophesytime = gettimeofday( &begin_time[62], (void *)0 );
#line 952 "is.c"
for( iteration=1; iteration<=10; iteration++ )

  {
    if( my_rank == 0 && 'A' != 'S' )
#line 954 "Instrumentation code"
( prophesytime = gettimeofday( &begin_time[60], (void *)0 ),
_printf_ReturnValue_ =
#line 954 "is.c"
          printf( "       %d\n", iteration )

#line 954 "Instrumentation code"
, prophesytime = gettimeofday( &exit_time, (void *)0 ),
prophesyruntime = exit_time.tv_sec - begin_time[60].tv_sec,
prophesyruntime += ( (double)( exit_time.tv_usec -
begin_time[60].tv_usec ) / 1000
000.0 ),
celero[60].runtime += prophesyruntime,
celero[60].psqrtime += prophesyruntime * prophesyruntime,
celero[60].count ++
, _printf_ReturnValue_)
#line 954 "is.c"
;

#line 955 "Instrumentation code"
( prophesytime = gettimeofday( &begin_time[61], (void *)0 ),
#line 955 "is.c"
          rank( iteration )
#line 955 "Instrumentation code"
, prophesytime = gettimeofday( &exit_time, (void *)0 ),
prophesyruntime = exit_time.tv_sec - begin_time[61].tv_sec,
prophesyruntime += ( (double)( exit_time.tv_usec -
begin_time[61].tv_usec ) / 1000
000.0 ),
celero[61].runtime += prophesyruntime,
celero[61].psqrtime += prophesyruntime * prophesyruntime,
celero[61].count ++
)
#line 955 "is.c"
;
  }
#line 956 "Instrumentation code"

prophesytime = gettimeofday( &exit_time, (void *)0 );
prophesyruntime = exit_time.tv_sec - begin_time[62].tv_sec;
prophesyruntime += ( (double)( exit_time.tv_usec -
begin_time[62].tv_usec ) / 1000
000.0 );
celero[62].runtime += prophesyruntime;
celero[62].psqrtime += prophesyruntime * prophesyruntime;
celero[62].count ++;

#line 956 "is.c"
```

This instrumentation is for all procedures and the outer loop to illustrate the instrumentation statement added to an application source code. For the main for-loop, its eventID is 62. Its instrumentation initializes the timer to measure the loop duration. At the end of the outer loop, it computes the duration of the loop and increments its timer, square of timer, and counter. For the function rank(), its eventID is 61. Its instrumentation code is similar to that for the outer loop as above.

3. CASE STUDIES: PERFORMANCE ANALYSIS AND COMPARISON

In this section, we use NAS CG and IS benchmarks to test the PAIDE system. For the serial CG benchmarks, we analyze the instrumentation overheads for two cases, Case I and Case II. Case I consists of instrumentation for procedures and outer loops and Case II consists of instrumentation for all procedures (that include the procedures in loops.) and outer loops. We analyze the two cases for different problem sizes and different number of processors. The following experimental results are obtained from SGI origin 2000. The SGI Origin2000 is in the Center for Parallel and Distributed Computing at Northwestern University. It is an eight-way symmetric multiprocessor. Each processor is a 64-bit chip running at 195 MHz capable of 390 Mflops (2 flops per cycle). The total main memory is 1GB.

3.1 NAS CG Benchmarks (Fortran version)

The NAS CG Benchmark [1] is an iterative algorithm for solving a linear system of equations, for which the linear system is represented as a sparse matrix. Table 1 shows the matrix orders and the number of iterations for problem size S, W, A and B.

Table 1. Number of iterations and serial execution times for problem size S, W, A and B

Problem size	Class S	Class W	Class A	Class B
Matrix order	1400	7000	14000	75000
No. Iterations	15	15	15	75
Original	0.86	10.89	47.41	2529.41
Case I	0.87	11.18	47.96	2538.53
Case II	0.88	11.27	48.04	2581.70

3.1.1 Serial NAS CG benchmark

With increasing the problem size from S to B, Table 1 shows the execution times of the original CG program, the instrumented program in Cases I and II for different problem sizes. Case I involves 33 instrumentation events, and Case II involves 39 instrumentation events. Table 2 shows the percent of the instrumentation overhead with increasing the problem size. The results indicate that the overhead is less than 3.4% in all cases. Further, for the large problem size with a large number of iterations, indicating a large number of times the PAIDE instrumentation code is executed, the overhead is less than 3%.

Table 2. Percent of Instrumentation overhead perturbation for serial CG

Problem size	Class S	Class W	Class A	Class B
Case I	1.15%	2.59%	1.15%	0.36%
Case II	2.27%	3.37%	1.31%	2.03%

3.1.2 Parallel NAS CG Benchmark with Class A

For the parallel CG benchmark with Class A for the problem size, Case I involves 48 instrumentation events, and Case II involves 82 instrumentation events. Figure 3 shows the execution times of the original CG program and the two cases with increasing number of processors from one to eight processors. The corresponding percentages are given in Table 3. The results also illustrate the overhead to be less than 3.4%, which is consistent with the sequential case.

Table 3. Percent of Instrumentation overhead for parallel CG with Class A

Processors	1	2	4	8
Case I	1.15%	1.27%	2.01%	1.91%
Case II	1.31%	2.14%	3.22%	3.15%

Table 4. Percent of instrumentation overhead perturbation for parallel IS with Class B

Processors	1	2	4	8
Case I	4.90%	0.43%	0.43%	0.22%
Case II	5.45%	0.44%	0.68%	0.81%

3.2 NAS Parallel IS Benchmarks (C version)

The IS benchmark [1] is parallel sort over small integers. It sorts N keys in parallel. The keys are generated by the sequential key generation algorithm and initially must be uniformly distributed in memory. Notice that the initial distribution of the keys can have a great impact on the performance of the benchmark.

For the parallel IS benchmark with Class B, it sorts 2^{25} keys in parallel. Case I involves 45 instrumentation events, and Case II involves 52 instrumentation events. Figure 4 shows the execution times of the original program, the instrumented program for the two cases with different number of processors. Table 4 shows the percent of the instrumentation overhead with increasing the number of processors. Again the percents are low and highly dependent on the initial distribution of memory.

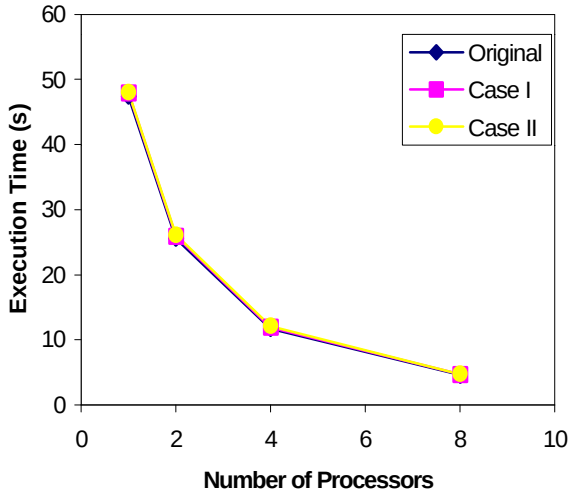

Figure 3. Execution times for parallel CG with Class A

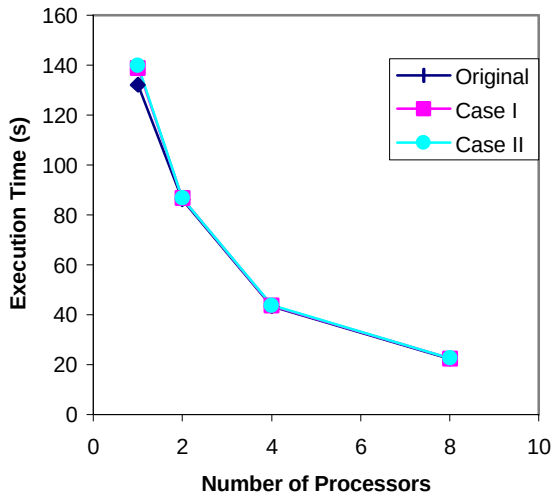

Figure 4. Execution times for parallel IS with Class B

In Table 3 and 4, the percent of the instrumentation overhead for each parallel execution is less than 3.4%. Especially, for the parallel IS benchmark with the large problem size Class B, the percent of the instrumentation overhead is less than 1%. The two case studies show that we basically meet the goal of the PAIDE: the minimal instrumentation overhead and code, and minimal perturbation of application programs. The PAIDE system is practical and more appropriate for parallel and distributed applications.

4. SUMMARY

In this paper, we present the design and implementation of the PAIDE system, for which the goal is to minimize instrumentation overhead or perturbation. The PAIDE automatically instruments application by inserting code to generate in-line first and second order statistics about the code segment. No I/O operations occur during the execution of the application; the PAIDE requires only one I/O write operation that occurs at the end of execution of each process. In addition to instrumenting the application, the PAIDE generates an events file (that maps the event identifiers with the lines of code in the application) and a control flow file. Further, unique identifiers are used to distinguish performance statistics for functions or routines that call similar code segments. The performance data can be used to generate models and aid with identifying bottlenecks and developing more efficient codes.

The experimental results using the PAIDE system with two of the NAS Parallel Benchmarks indicates that the PAIDE achieves its original goal of minimal overhead. In particular, the overhead for the PAIDE was less than 3.4% for CG and less than 1% for parallel IS. This is significant and it indicates that the PAIDE does not perturb the performance data, and is appropriate for parallel and distributed applications that take a long time to execute. Some works for the PAIDE system will be further done, such as the support for C++ and HPF programs, etc.

ACKNOWLEDGEMENTS

This research work was supported in part by the National Science Foundation under NSF grant EIA-9974960 and a NASA grant.

REFERENCES

[1] D. Bailey, T. Harris, et al., *The NAS Parallel Benchmarks*, Tech. Report NAS-95-020, Dec. 1995. See also http://science.nas. nasa.gov/Software/NPB/.

[2] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek and V. Sunderam, *PVM: Parallel Virtual Machine – A Users' Guide and Tutorial for Networked Parallel Computing*, the MIT Press, 1994.

[3] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, the MIT Press, 1994.

[4] A. Goldberg and J. Hennessy, Performance debugging shared-memory multiprocessor programs with MTOOL, in *Proc. of Supercomputing'91*.

[5] G. Hansen, C. Linthicum, and G. Brooks, Experience with a performance analyzer for multithreaded application, in *Proc. of supercomputing'90*.

[6] E. Karrels and E. Lusk, *Performance Analysis of MPI programs*, available from the URL: ftp://info.mcs.anl.gov/pub/mpi/misc/heath.ps.

[7] B. Miller, J. Cargille, et al., *The Paradyn Parallel Performance Measurement Tools*, http://www.cs.wisc.edu/~paradyn/papers.html.

[8] B. Miller, D. Callaghan, et al., The Paradyn parallel performance measurement tools, *IEEE Computer*, 28(11), 1995.

[9] B. Miller, M. Clark, et al., IPS-2: The second generation of a parallel program measurement system, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 2, 1990, 206-217.

[10] D. Reed, R. Olson, et al., Scalable performance environments for parallel systems, in *Proc. of the 6th Distributed Memory Computing Conference*, IEEE Computer Society Press, 1991, 562-569.

[11] L. De Rose, Y. Zhang, M. Pantano, and S. Whitmore, *SvPablo Guide*, Pablo Research Group, April 2000.

[12] V. Taylor, X. Wu, J. Geisler, X. Li, Z. Lan, R. Stevens, M. Hereld, and I. Judson, Prophesy: An infrastructure for analyzing and modeling the performance of parallel and distributed applications, in *Proc. Of the 9th IEEE International Symposium on High Performance Distributed Computing (short paper)*, IEEE Computer Society Press, August 2000.

[13] X. Wu, V. Taylor, J. Geisler, X. Li, Z. Lan, R. Stevens, M. Hereld, and I. Judson, Design and development of Prophesy performance database for distributed scientific applications, in *Proc. 10th SIAM Conference on Parallel Processing for Scientific Computing*, March 2001.